# Detección de interleavings no serializables usando contadores de hardware

Fernando Emmanuel Frati[1, 2], Katzalin Olcoz Herrero[3], Luis Piñuel Moreno[3], Marcelo Naiouf[1], Armando De Giusti[1,2]

[1] Instituto de Investigación en Informática LIDI (III-LIDI),
Facultad de Informática, Universidad Nacional de La Plata, Argentina
{fefrati, mnaiouf, degiusti}@lidi.info.unlp.edu.ar
[2] Consejo Nacional de Investigaciones Científicas y Técnicas
Facultad de Informática, Universidad Nacional de La Plata, Argentina
[3] Departamento de Arquitectura de Computadores y Automática,
Facultad de Informática, Universidad Complutense de Madrid, España
{katzalin, lpinuel}@dacya.ucm.es

**Resumen.** El análisis de interleavings es una técnica que permite detectar violaciones de atomicidad en programas de memoria compartida. Los errores de concurrencia están fuertemente ligados al no determinismo presente en la ejecución de las aplicaciones, por lo que se considera necesario contar con herramientas que permitan detectar su presencia en entornos de producción. Lamentablemente, los algoritmos que implementan esta técnica suelen ser muy costosos en tiempo de ejecución, restringiendo su uso a etapas de prueba del software. Este trabajo muestra cómo se pueden utilizar los contadores hardware presentes en las arquitecturas de los procesadores actuales para detectar la ocurrencia de interleavings no serializables. Esta optimización permitirá disminuir el overhead que introducen las herramientas de detección de errores de concurrencia.

**Palabras clave:** arquitecturas paralelas, programa concurrente, error de concurrencia, detección de errores, depuración, contadores de hardware.

## 1 Introducción

En todo programa concurrente es el programador quien debe especificar de qué manera se sincronizarán los procesos del programa. En función del modelo de comunicación empleado, existen diferentes métodos para establecer la sincronización. Por ejemplo, en un modelo de memoria compartida es común la utilización de semáforos o monitores y en uno de memoria distribuida se suele utilizar pasaje de mensajes. Los errores de concurrencia aparecen cuando el programador se equivoca en la utilización de alguno de estos métodos, provocando condiciones de carrera, deadlocks o violaciones de atomicidad.
Una característica que hace a los errores de concurrencia particularmente difíciles de detectar, es que sólo se manifiestan bajo determinadas condiciones de ejecución, dependiendo principalmente del no determinismo presente en el orden de ejecución de los procesos. Si estos errores no se manifiestan durante el período de prueba, el

programa llegará a formar parte de los sistemas en producción para los que fue pensado volviéndolos vulnerables.

Estos errores, las causas que los producen y formas para evitarlos han sido estudiados ampliamente por la comunidad científica. En 1978 Lamport estableció el concepto de orden parcial entre segmentos de procesos [1], al que llamó relación *happens before*, y ha sido utilizado para la construcción de herramientas de detección de condiciones de carrera. En 1997 se propuso Lockset [2], un método diferente para detectar condiciones de carrera que controla que todas las variables compartidas están protegidas en el momento de accederlas. Los deadlocks son otro tipo de error común presente en programas concurrentes. En 1972 Holt [3] propuso un modelo capaz de detectar condiciones de deadlock por recursos compartidos (de hecho esta técnica es la utilizada por los sistemas operativos para la gestión de sus recursos). Por último, con la llegada de los multiprocesadores a las computadoras convencionales, las violaciones de atomicidad han comenzado a ocupar un rol protagónico por tratarse de un caso más general que los anteriores. Shan Lu postuló en 2006 [4] un análisis del orden en que varios threads acceden a memoria para detectar violaciones de atomicidad. La técnica consiste en clasificar los *interleavings*[1] que ocurren en el programa en serializables y no serializables, donde los últimos pueden ser violaciones de atomicidad (este tema se aborda en una sección posterior del trabajo). Las violaciones de atomicidad también han sido tratadas por otros autores [5, 6].

Cualquiera de estos errores se pueden encontrar con frecuencia en programas ampliamente utilizados, como por ejemplo Apache, MySQL o Mozilla Firefox [7]. Debido al no determinismo presente en las ejecuciones paralelas, sería muy útil contar con herramientas que permitan monitorizar aplicaciones en entornos en producción.

Lamentablemente el overhead que introducen los algoritmos de detección constituye un factor determinante en la viabilidad de su utilización. Por este motivo las propuestas actuales tienden a incluir una versión de sus algoritmos que utiliza extensiones hardware para acelerarlos, y disminuir de esta manera el impacto de la instrumentación. Estas extensiones implican cambios a la arquitectura de las máquinas donde se ejecutan (por ejemplo agregar bits a la línea de cache).

Este trabajo propone un enfoque diferente: utilizar la información disponible a través de los contadores hardware del procesador para elegir cuándo ejecutar los algoritmos de detección. Experimentos anteriores [8] indican que gran parte del overhead causado por la herramienta de monitorización se debe a la instrumentación de regiones de código seguras. Por ejemplo, en el caso de una multiplicación de matrices, sólo el 15% de las instrucciones ejecutan interleavings no serializables. Una técnica similar fue empleada por Greathouse en 2011 [9] para optimizar una herramienta de detección de condiciones de carrera de Intel llamada Inspector XE. Sin embargo, dicha herramienta sólo es capaz de detectar condiciones de carrera y utiliza algoritmos basados en happens-before y lockset. El objetivo de este trabajo es determinar la viabilidad de la utilización de los contadores en la detección de interleavings no serializables, lo que permitirá optimizar técnicas de detección de

1 Se optó por el término en inglés porque es el utilizado en la literatura sobre el tema. Una posible traducción podría ser accesos a memoria intercalados o entrelazados.

violaciones de atomicidad en entornos de memoria compartida. La hipótesis es que se reducirá el overhead al restringir la instrumentación a aquellos accesos que efectivamente son no serializables.
El artículo se organiza de la siguiente manera: en la sección 2 se describe la técnica de análisis de interleavings que se busca optimizar. La sección 3 presenta los contadores hardware y proporciona un análisis sobre cómo podrían servir para la detección de interleavings no serializables. La sección 4 describe el entorno experimental, la metodología empleada y los algoritmos utilizados para implementar la técnica propuesta en este trabajo. La sección 5 muestra los resultados de los experimentos. Finalmente en la sección 6 las conclusiones.

## 2 Análisis de Interleavings

Una manera de ver los errores consiste en analizar los patrones de acceso (lecturas y escrituras) que varios procesos hacen a cada dirección de memoria. Cuando entre dos accesos de un mismo thread ocurre un acceso de un thread diferente a la misma variable, se lo denomina interleaving. Dependiendo de la configuración del interleaving, el resultado podría ser un error de concurrencia. La Tabla 1 resume las configuraciones posibles. Los accesos del primer thread se encuentran alineados a la izquierda, mientras que el acceso entrelazado del segundo thread se encuentra desplazado a la derecha.

| Caso | Interleaving |
|---|---|
| 0 | read<br>  read<br>read |
| 1 | write<br>  read<br>read |
| 2 | read<br>  write<br>read |
| 3 | write<br>  write<br>read |

| Caso | Interleaving |
|---|---|
| 4 | read<br>  read<br>write |
| 5 | write<br>  read<br>write |
| 6 | read<br>  write<br>write |
| 7 | write<br>  write<br>write |

**Tabla 1**: cada caso muestra una configuración diferente de interleaving. Los accesos corresponden a operaciones entrelazadas de lectura/escritura a una misma dirección de memoria entre dos threads.

De los ocho casos posibles, cuatro pueden ser serializados. Esto significa que la ocurrencia del interleaving no altera la percepción de los procesos involucrados sobre la región de memoria a la que acceden, y por lo tanto se consideran seguros. Los interleavings 0, 1, 4 y 7 son seguros porque su ocurrencia produce el mismo efecto que si no hubiese ocurrido. En cambio los casos 2, 3, 5 y 6 son interleavings que

pueden haber sido causados por un error de concurrencia.
Las propuestas de este tipo buscan la ocurrencia de interleavings no serializables, y cuando los detectan emiten una alerta o inician un procedimiento correctivo (inmediatamente después de que el error se manifiesta), llegando incluso a determinar cuáles son las instrucciones que provocaron el error.
La instrumentación necesaria para realizar este tipo de análisis introduce un elevado overhead en el tiempo de ejecución de las aplicaciones monitorizadas. Tal como se mencionó en la introducción, los algoritmos para implementar esta técnica incluyen una versión que utiliza extensiones hardware para acelerarlos, y disminuir de esta manera el impacto de la instrumentación. La Tabla 2 muestra los resultados de ejecutar las versiones software y hardware de AVIO sobre las aplicaciones de la suite de benchmarks SPLASH-2 [4]. Lamentablemente estas extensiones implican cambios a la arquitectura de las máquinas donde se ejecutan (por ejemplo agregar bits a la línea de cache), volviendo inviable su aplicación en entornos de producción reales.

| Benchmark | Overhead introducido por la herramienta de detección de bugs | |
|---|---|---|
| | AVIO (Hardware) | AVIO (Software) |
| fft | 0,5 % | 42X |
| fmm | 0,4 % | 19X |
| lu | 0,4 % | 23X |
| radix | 0,4 % | 15X |
| Promedio | 0,4 % | 25X |

**Tabla 2**: Comparación de overhead entre herramientas de detección de bugs sobre la suite de benchmarks SPLASH-2. La X en la tabla representa el tiempo de la aplicación monitorizada sin instrumentación.

La alternativa propuesta en este trabajo consiste en utilizar la información que proveen los contadores hardware sobre lo que está ocurriendo en el procesador para distinguir qué accesos necesitan ser monitorizados y cuáles no. Como se mencionó en la introducción, gran parte del tiempo invertido en monitorizar accesos se puede reducir si se restringe sólo a las regiones de código que son inseguras (en el caso de la multiplicación de matrices es del 15%). Esta optimización permitirá reducir considerablemente el overhead de la versión software.

## 3 Contadores de Hardware

Todos los procesadores actuales poseen un conjunto de registros especiales denominados contadores de hardware [10]. Estos registros se pueden programar para contar el número de veces que ocurre un evento dentro del procesador durante la ejecución de una aplicación. Tales eventos pueden proveer información sobre

diferentes aspectos de la ejecución de un programa (por ejemplo el número de instrucciones ejecutadas, la cantidad de fallos cache en L1 o cuántas operaciones en punto flotante se ejecutaron). Esta información pretende ayudar al programador a comprender mejor el desempeño de sus algoritmos y optimizar su programa. Estos eventos se encuentran pobremente documentados, y suelen variar entre diferentes arquitecturas. Los fabricantes de procesadores habitualmente ofrecen un manual donde indican los eventos y una breve descripción de lo que cuentan.
El objetivo a cumplir es encontrar eventos que permitan detectar si se están ejecutando interleavings no serializables. Para ello se analizó el uso de la cache y los cambios de estado de coherencia que producen las operaciones de lectura/escritura sobre ella. Se parte de un modelo de interacción entre dos threads que comparten variables. La Tabla 3 resume los cuatro casos posibles.

| Caso | Patrón |
|---|---|
| 0 | read<br>read |
| 1 | read<br>write |

| Caso | Patrón |
|---|---|
| 2 | write<br>read |
| 3 | write<br>write |

**Tabla 3**: cada caso muestra un patrón de acceso diferente a una misma dirección de memoria entre dos threads.

En todos los casos el evento debe ocurrir con cada read o write. En los protocolos de coherencia basados en MESI existe un estado “Compartido”(S) que indica que la línea de cache ha sido leída por más de un procesador. Los eventos candidatos son aquellos que indican una transición hacia o desde este estado. Cada patrón de acceso sirve para detectar diferentes interleavings. La Tabla 4 muestra la relación existente entre estos. Como puede observarse, los patrones de acceso útiles para el objetivo de este trabajo son el 1 y el 2, ya que permiten seguir el rastro de mayor cantidad de interleavings no serializables (2, 3, 5 y 6).

# 4 Trabajo Experimental

**Herramienta de monitorización.** Para realizar los experimentos se desarrolló una herramienta en el framework de instrumentación dinámica binaria Pin [11]. El objetivo es determinar si se puede detectar un patrón de acceso a partir de la cuenta de un evento. La herramienta se sitúa entre el hardware y la aplicación monitorizada y está diseñada para analizar programas paralelizados con Pthreads. Esta se encarga de configurar e iniciar la cuenta del evento en el momento en que se crea el thread. Para asegurar que cada thread se ejecute en un núcleo diferente también configura su afinidad mediante una llamada a la función sched_setaffinity. Al finalizar el thread, se guarda la cuenta del evento en una variable. Luego cuando finaliza la aplicación monitorizada, la herramienta informa la cuenta total de eventos para cada thread. La principal ventaja de esta técnica es que no requiere modificar el código fuente de la aplicación monitorizada, lo que permite aplicarla sobre cualquier aplicación en tiempo

| Patrón de acceso / Interleaving | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|
| | | read read read | write read read | read write read | write write read | read read write | write read write | read write write | write write write |
| **0** | read read | SI | SI | - | - | SI | - | - | - |
| **1** | read write | - | - | SI | - | SI | SI | SI | - |
| **2** | write read | - | SI | SI | SI | - | SI | - | - |
| **3** | write write | - | - | - | SI | - | - | SI | SI |

**Tabla 4**: cada celda muestra si el patrón de acceso puede indicar la ocurrencia de el interleaving.

de ejecución.

**Micro-benchmarks.** Para verificar que los eventos candidatos se pueden usar para distinguir entre diferentes patrones de acceso, se diseñaron los micro-benchmarks de la Figura 1.

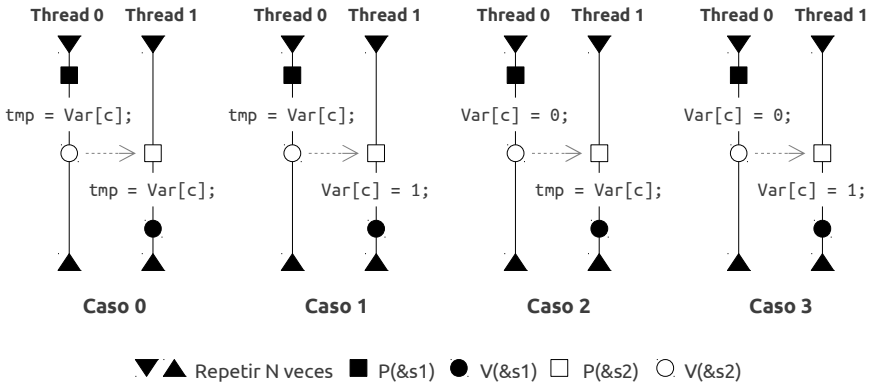


**Figura 1**: micro-benchmarks para simular los cuatro patrones de acceso.

Cada caso corresponde a un programa desarrollado con la librería Pthreads, en que se repite la interacción simulada entre dos threads *N* veces (*N* es un parámetro del programa). Cada thread posee una variable local llamada *tmp* utilizada para simular la lectura. La variable compartida está representada por un arreglo *Var* de *N* elementos, inicializado por el thread principal del programa. Para evitar que alguna optimización sobre la librería Pthreads en el uso de semáforos pueda entorpecer el análisis de los resultados, se optó por implementar las funciones *P()* y *V()* con variables compartidas. Las variables *s1* y *s2* cumplen la función de semáforos binarios y se

encuentran inicializadas en 1 y 0 respectivamente. En la Figura 2 se puede ver el pseudocódigo de estas operaciones. Se debe notar que en esta implementación de *P()* podría ocurrir una violación de atomicidad si entre el *while* y el decremento de *s* ocurriera un *V()*. Esta situación podría ocurrir si el thread 1 iterara lo suficientemente rápido como para ejecutar un segundo *V(&s1)* antes que el thread 0 completara el decremento de esta variable. Sin embargo, como antes de que el thread 1 ejecute el siguiente *V(&s1)* deberá esperar en *P(&s2)* a que el thread 0 ejecute *V(&s2)*, la violación no ocurrirá nunca.

Debido a que los eventos cambian el estado de la línea completa de cache, se diseñó una estructura que ocupa la línea completa para los elementos del arreglo *Var* y para *s1* y *s2*. Esto permite analizar los cambios de estado asociados a las operaciones de lectura/escritura sobre estas variables.

**Plataforma.** Los experimentos se realizaron sobre una máquina con dos procesadores Xeon X5670, cada uno de los cuales posee 6 núcleos con HT. El sistema operativo utilizado es un Debian wheezy/sid x86_64 GNU/Linux con un kernel 3.2.0. El compilador utilizado es gcc, versión 4.7.0. No se usaron opciones de optimización ya que éstas podrían modificar los códigos que se quieren analizar. La versión utilizada de pin es la 2.11. El acceso a los contadores se hace a través de perf_event, el sistema incluido en los kernels de linux a partir de la versión 2.6.33. Los micro-benchmarks se desarrollaron usando la librería pthreads. La micro-arquitectura de estos procesadores, llamada *Westmere,* corresponde a la micro-arquitectura *Nehalem* con tecnología de 32nm. El procesador utilizado tiene tres niveles de cache: L1 y L2 privadas de cada núcleo y L3 compartida entre todos los núcleos del mismo procesador físico. Las cache L1 y L2 son no inclusivas, mientras que la L3 sí lo es. La L3 tiene un flag adicional que permite saber cuál fue el último núcleo en actualizar una línea [12]. El protocolo de coherencia está basado en MESI, pero se agrega un quinto estado "Forward" (F), para indicar que la línea fue enviada de un socket al otro. En este esquema una línea que ha sido leída por varios núcleos estará en estado S para todas las copias y en todos los niveles (L1, L2 y L3). Una escritura cambiará el estado de la línea de cache a M en las caches del procesador que hace la escritura y en L3. Este cambio se propagará también a las copias de la línea en las cache que la tengan en los otros núcleos, cambiando su estado a I [13].

**Selección de eventos.** En el Manual para el desarrollador de software para las

```
P(linea_cache *s){
  while(*s <= 0);
  *s--;
}

V(linea_cache *s){
  *s++;
}
```

**Figura 2**: funciones *P()* y *V()* utilizadas para sincronizar los threads. Ambas operaciones implican al menos una lectura seguida de una escritura sobre la variable apuntada por *s*.

arquitecturas de Intel [14] se encuentra la lista de eventos disponibles para la micro-arquitectura *Westmere*. Teniendo en cuenta que el evento tiene que detectar el patrón 1 o 2, se seleccionó por su descripción como evento candidato MEM_UNCORE_RETIRED:LOCAL_HITM. La descripción de este evento dice que cuenta instrucciones *loads* que aciertan en datos en estado modificado en un *sibling core* (otro núcleo en el mismo procesador). También indica que es un evento preciso. Se debe observar que el evento ocurre luego del *read* del patrón de acceso 2. Aunque no se encontró un evento que detecte el patrón de acceso 1, hay que considerar que los interleavings no suelen ocurrir de manera aislada como en los micro-benchmarks diseñados para este trabajo. La optimización propuesta de activar/ desactivar la herramienta de monitorización en función de este evento permitirá detectar tres de los cuatro interleavings no serializables. La ocurrencia del patrón de acceso 2 es un claro indicador de se están compartiendo datos entre los threads y, por lo tanto, es probable que la herramienta se encuentre activa cuando el interleaving caso 6 ocurra la mayoría de las veces. Esta hipótesis será evaluada en futuros experimentos.

## 5 Resultados

Para disminuir el margen de error en las mediciones, cada prueba se realizó diez veces y se promediaron los resultados. Cada patrón de acceso se ejecutó con diferentes tamaños de *N* a los efectos de estudiar el comportamiento cuando el tamaño del problema varía. El experimento fue diseñado para demostrar que el evento seleccionado puede detectar cuando ocurren patrones de acceso del caso 2, lo cuál es un indicador de interleavings no serializables de los casos 2, 3 y 5. Debido a que los micro-benchmarks diseñados usan una variable compartida para sincronizar threads, se espera obtener la ocurrencia de un evento por cada iteración del bucle (se debe recordar que el evento ocurre con cada lectura después de escritura en la cache de otro core). La única diferencia debe ocurrir con el caso 2, donde para el thread 1 debe ocurrir un evento extra por cada iteración debido a la operación de lectura después de escritura en la variable *Var[c]*. Los resultados de los experimentos se pueden apreciar en la Tabla 5.

Debajo de cada configuración de experimento se puede observar con fondo gris el cociente entre el número de ocurrencias del evento y el tamaño del problema. Los resultados se redondearon a dos decimales. Se debe notar que cada patrón de acceso presenta el mismo comportamiento independientemente del tamaño del problema. En todos los casos, por cada iteración del bucle del micro-benchmark se observa que se registró aproximadamente una ocurrencia del evento. Esto se puede explicar a partir de las Figuras 1 y 2. Para conseguir que las operaciones de lectura/escritura ocurran en el orden adecuado, los threads se sincronizan usando dos variables compartidas, *s1* y *s2*. Por ejemplo el thread 1 se encuentra realizando lecturas a *s2* hasta que el thread 0 incremente su valor (se debe observar la flecha gris en la Figura 1).

Debido a que la política de escritura es *write-allocate*, el procesador que está ejecutando el thread 0 trae a su cache la variable *s2* y la actualiza, dejando su estado de coherencia en M (modificado). Luego el thread 1 realiza una lectura que alcanza la

**MEM_UNCORE_RETIRED:LOCAL_HITM**

| | Patrón 0 | | Patrón 1 | | Patrón 2 | | Patrón 3 | |
|---|---|---|---|---|---|---|---|---|
| | **Thr0** | **Thr1** | **Thr0** | **Thr1** | **Thr0** | **Thr1** | **Thr0** | **Thr1** |
| **$10^4$** | 11807,7 | 11138,1 | 11635,8 | 11058,3 | 12420,8 | 20114,6 | 11654,7 | 10791,3 |
| | 1,18 | 1,11 | 1,16 | 1,11 | 1,24 | 2,01 | 1,17 | 1,08 |
| **$10^5$** | 101987,2 | 98839,6 | 94526,7 | 98601 | 97233,9 | 192174,5 | 95997,3 | 95941,7 |
| | 1,02 | 0,99 | 0,95 | 0,99 | 0,97 | 1,92 | 0,96 | 0,96 |
| **$10^6$** | 1009867,8 | 980244,2 | 926031,5 | 974063,8 | 936528,9 | 1904729,5 | 920952,7 | 894380,1 |
| | 1,01 | 0,98 | 0,93 | 0,97 | 0,94 | 1,90 | 0,92 | 0,89 |

**Tabla 5**: las celdas contienen la cuenta del evento de cada patrón de acceso entre dos threads para diferentes tamaños de problema

línea en estado M en la cache del otro núcleo, disparando el evento. Lo mismo ocurre con *s1* para el thread 0. La única diferencia entre los patrones de acceso es con el caso 2, donde se puede observar que el thread 1 registra un evento adicional por cada iteración. Esto se explica por la lectura después de escritura realizada a la variable *Var[c]*, lo que demuestra que el evento sirve para diferenciar el patrón 2 de los demás.

# 6 Conclusiones

Este trabajo muestra un modelo para la utilización de contadores hardware en la detección de interleavings no serializables. A través del uso de técnicas de instrumentación, se consiguió establecer una relación entre los patrones de acceso a memoria, el tamaño del problema y el evento propuesto. De los experimentos realizados se concluye que se pueden usar los contadores para detectar la ejecución de al menos tres de los cuatro casos de interleavings no serializables.

Actualmente se está trabajando en utilizar esta información para detectar los instantes en que éstos comienzan a ocurrir. El objetivo es desarrollar una herramienta de detección de violaciones de atomicidad que se active sólo en aquellos segmentos de programa que ejecutan este tipo de interleavings, con lo que se espera conseguir una mejora en los tiempos de ejecución de los algoritmos de detección.

Nuestros resultados previos muestran que gran parte del tiempo invertido en monitorizar accesos se puede reducir restringiendo la instrumentación sólo a las regiones de código que son inseguras (en el caso de la multiplicación de matrices es del 15%). Recordando que AVIO introduce una penalización aproximada de 24X, se considera que esta optimización permitirá reducir considerablemente el overhead de la versión software.

## 7 Referencias


1. Lamport, L.: Time, clocks, and the ordering of events in a distributed system, (1978).
2. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: a dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. 15, 391–411 (1997).
3. Holt, R.C.: Some Deadlock Properties of Computer Systems. ACM Comput. Surv. 4, 179–196 (1972).
4. Lu, S., Tucek, J., Qin, F., Zhou, Y.: AVIO: detecting atomicity violations via access interleaving invariants. SIGPLAN Not. 41, 37–48 (2006).
5. Lucia, B., Devietti, J., Strauss, K., Ceze, L.: Atom-Aid: Detecting and Surviving Atomicity Violations. ISCA '08: Proceedings of the 35th Annual International Symposium on Computer Architecture. pp. 277–288. IEEE Computer Society, Washington, DC, USA (2008).
6. Lucia, B., Ceze, L., Strauss, K.: ColorSafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations. SIGARCH Comput. Archit. News. 38, 222–233 (2010).
7. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. SIGARCH Comput. Archit. News. 36, 329–339 (2008).
8. Frati, F.E., Olcoz Herrero, K., Moreno, L.P., Montezanti, D.M., Naiouf, M., De Giusti, A.: Optimización de herramientas de monitoreo de errores de concurrencia a través de contadores de hardware. Proceedings del XVII Congreso Argentino de Ciencia de la Computación. p. 10. Red UNCI, La Plata (2011).
9. Greathouse, J.L., Ma, Z., Frank, M.I., Peri, R., Austin, T.: Demand-Driven Software Race Detection using Hardware Performance Counters. (2011).
10. Sprunt, B.: The basics of performance-monitoring hardware. IEEE Micro. 22, 64–71 (2002).
11. Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. pp. 190–200. ACM, New York, NY, USA (2005).
12. Intel® 64 and IA-32 Architectures Optimization Reference Manual. Intel Corporation (2012).
13. Levinthal, D.: Performance Analysis Guide for Intel® CoreTM i7 Processor and Intel® XeonTM 5500 processors. Intel Corporation (2009).
14. Intel® 64 and IA-32 Architectures Software Developer's Manual. Intel Corporation (2012).